In programming, the algorithms we use to solve complex problems in a systematic and controlled way are designed on the basis of two approaches that is : **Top-down** and **Bottom-up approach**. The ideology behind top-down approach is, a bigger problem is divided into some smaller sub-problems called modules, these modules are then solved individually and then integrated together to get the complete solution to the problem.

Top-Down Approach

The basic idea in top-down approach is to break a complex algorithm or a problem into smaller segments called **modules**, this process is also called as **modularization**. The modules are further decomposed until there is no space left for breaking the modules without hampering the originality. The uniqueness of the problem must be retained and preserved. The decomposition of the modules is restricted after achieving a certain level of modularity. The top-down way of solving a program is step-by-step process of breaking down the problem into chunks for organising and solving the sole problem. The C- programming language uses the top-down approach of solving a problem in which the flow of control is in the downward direction.

C programming language supports this approach for developing projects. It is always good idea that decomposing solution into modules in a hierarchal manner. In this approach, first we develop the main module and then the next level modules are developed. This procedure is continued until all the modules are developed.

Advantages of top-down approach:

1. In this approach, first, we develop and test most important module.
2. This approach is easy to see the progress of the project by developer or customer.
3. Using this approach, we can utilize computer resources in a proper manner according to the project.
4. Testing and debugging is easier and efficient.
5. In this approach, project implementation is smoother and shorter.
6. This approach is good for detecting and correcting time delays.

Modular Programming

Modular programming is the process of subdividing a computer program into separate sub-programs. A module is a separate software component. It can often be used in a variety of applications and functions with other components of the system. Some programs might have thousands or millions of lines and to manage such programs it becomes quite difficult as there might be too many of syntax errors or logical errors present in the program, so to manage such type of programs concept of modular programming approached. Each sub-module contains something necessary to execute only one aspect of the desired functionality. Modular programming emphasis on breaking of large programs into small problems to increase the maintainability, readability of the code and to make the program handy to make any changes in future or to correct the errors.

Advantages of using Modular Programming :

1. **Ease of Use**: This approach allows simplicity, as rather than focusing on the entire thousands and millions of lines code in one go we can access it in the form of modules. This allows ease in debugging the code and prone to less error.
2. **Reusability**: It allows the user to reuse the functionality with a different interface without typing the whole program again.

3. **Ease of Maintenance**: It helps in less collision at the time of working on modules, helping a team to work with proper collaboration while working on a large application.

Concept of Modularization

One of the most important concepts of programming is the ability to group some lines of code into a unit that can be included in our program. The original wording for this was a sub-program. Other names include: **macro**, **sub-routine**, **procedure**, **module** and **function**. Functions are important because they allow us to take large complicated programs and to divide them into smaller manageable pieces. Because the function is a smaller piece of the overall program, we can concentrate on what we want it to do and test it to make sure it works properly. Generally, functions fall into two categories:

1. **Program Control** – Functions used to simply sub-divide and control the program. These functions are unique to the program being written. Other programs may use similar functions, maybe even functions with the same name, but the content of the functions are almost always very different.

Specific Task – Functions designed to be used with several programs. These functions perform a specific task and thus are usable in many different programs because the other programs also need to do the specific task. Specific task functions are sometimes referred to as building blocks. Because they are already coded and tested, we can use them with confidence to more efficiently write a large program.

Standard Library Functions in C

Standard Library Functions are basically the **inbuilt functions** which are grouped together and placed in a common place called library. Each library function in C performs specific operation.

Every C program has at least one function, that is, the **main()** function. The **main()** function is also a **standard library function in C** since it is inbuilt and conveys a specific meaning to the C compiler. We can make use of these library functions to get the pre-defined output instead of writing our own code to get those outputs. These library functions are created by the persons who designed and created C compilers. All C standard library functions are declared in many header files which are saved as **file_name.h**. Actually, function declaration, definition for macros are given in all header files. We can include these header files in our C program using **“#include<file_name.h>”** command to make use of the functions those are declared in the header files. When we include header files in our C program using **“#include<filename.h>”** command, all C code of the header files are included in C program. Then, this C program is compiled by compiler and executed.

Significance of Standard Library Functions in C

1. **Usability :** Standard library functions allow the programmer to use the pre-existing codes available in the C compiler without the need for the user to define his own code by deriving the logic to perform certain basic functions.
2. **Flexibility :** A wide variety of programs can be made by the programmer by making slight modifications while using the standard library functions in C.
3. **User-friendly syntax :** It is easy to grasp and use the syntax of functions.
4. **Optimization and Reliability :** All the standard library functions in C have been tested multiple times in order to generate the optimal output with maximum efficiency making it reliable to use.
5. **Time-saving : I**nstead of writing numerous lines of codes, these functions help the programmer to save time by simply using the pre-existing functions.

6. **Portability :** Standard library functions are available in the C compiler irrespective of the device you are working on. These functions connote the same meaning and hence serve the same purpose regardless of the operating system or programming environment.

Header Files in C

In order to access the standard library functions in C, certain header files need to be included before writing the body of the program.

A list of header files associated with some of the standard library functions in C:

1. <stdio.h> :- This is standard input/output header file used to perform input/output operations like printf() and scanf().
2. <conio.h> :- This console input/output header file used to perform console input and console output operations like clrscr() to clear the screen and getch() to get the character from the keyboard.
3. <stdlib.h> :- This is standard library file used to perform standard utility functions like dynamic memory allocation using functions such as malloc() and calloc().
4. <string.h> :- This is string header file used to perform string manipulation operations like strlen and strcpy.
5. <math.h> :- This is a math header file used to perform mathematical operations like sqrt() and pow() to obtain the square root and the power of a number respectively.
6. <ctype.h> :- This is a character type header file used to perform character type functions like isaplha() and isdigit() to find whether the given character is an alphabet or a digit. respectively.

```
/* Program to demonstrate the use of library functions */

#include<stdio.h>
#include<ctype.h>
#include<math.h>
void main()
{
  int i = -10, e = 2, d = 10; /* Variables Defining and Assign values */
  float rad = 1.43;
  double d1 = 3.0, d2 = 4.0;
  printf("%d\n", abs(i));
  printf("%f\n", sin(rad));
  printf("%f\n", cos(rad));
  printf("%f\n", exp(e));
  printf("%d\n", log(d));
  printf("%f\n", pow(d1, d2));
}
```

OUTPUT :

```
10
0.990105
0.140332
7.389056
-1145744106
81.000000
```

FUNCTIONS

A function is a self contained block of statements that performs some specific ,well defined task.

Types of functions:- Functions are of two types:-

1. **Library functions :-** These are the pre-defined functions and are stored in the library files. Such functions do not require to be created. They are simply called from the library files and are used. In order to use the library functions, the library files containing the definition of these functions, needs to be included in the program.
   For eg. In order to use printf() , scanf() functions , we need to include the stdio.h .
   In order to use clrscr() and getch() functions we need to include conio.h
   In order to use the mathematical functions we need to include math.h
   In order to use the string library we need to include string.h

2. **User-defined functions:-** These functions are created by the users for performing some specific task. These functions have the following three components :-

i. Function declaration
ii. Function definition
iii. Function calling

Function declaration:- Function declaration declares the function. It tells the compiler about :-

i. **Return type of the function: -** The return type of the function is any valid data type. If the function is not returning any value then **VOID** is written as the return type.
ii. **Name of the function: -** The name of the function is any valid identifier.
iii. **Arguments of the function:-** Function declaration tells about the number and type of the arguments received by the function.
   Function declaration tells the compiler that a function with the specified name and arguments is defined later in the program.
   The **general syntax** of the function declaration is as follows:-
   Return_type func_name(type1 arg1,type2 arg2,........);
   Eg. :-
   ```
   int sum(int ,int);
   void cube(int x);
   ```

Function definition:- The function definition consists of the whole description and code of the function. It tells about what the function is actually doing. A function definition consists of two parts:-

a) **A function header:-** The first line of the function definition is known as function header. It contains the return type of the function, function name and the formal parameters of the function. The return type of the function denotes the type of the value that will be returned by the function. The return type is optional and if omitted, is assumed to be int
   by default. A function can either return a value or no value. If the function is not going to return any value then void as written as the return type.
b) **A function body:-**The function body consists of the declaration of variables and the C statements followed by an optional return statement.
   The general syntax of the function definition is as follows:-
   ```
   Return_type  func_name(type1 arg1,type2 arg2,.......)
   {
   ```

```
                Local variables declaration;
                Statement1;
                Statement2;
                .
                 .
                          .
                 StatementN;
                 return (expression);
               }
```

Function calling:- It includes the calling of the function. A function is called by simply writing its name and the list of arguments inside the parenthesis.

The general syntax of the function call is as follows:-

Func_name(arg1,arg2,arg3,.......);

We give the following program to understand the concept of the function:-

/*program to find the sum of two numbers using the function*/

```
#include<stdio.h>
#include <conio.h>
void main()                                     /* main() function*/
{
int x,y,sum;                                    /*declaring the variables*/
clrscr();
int add(int,int);                                /* function declaration*/
printf("enter the two numbers");
scanf("%d%d",&x,&y);
sum=add(x,y);                                   /*function calling*/
printf("the sum of %d and %d  =%d",x,y,sum);
getch();
}
int add(int a,int b)                            /*function definition*/
{
int c;
c=a+b;
return c;                  /* the value of c is returned to the main() function.*/
}
```

Function Prototype

A function prototype declares the function name, its parameters, and its return type to the rest of the program prior to the function's actual declaration. It doesn't contain function body.

Function prototype tells compiler about number of parameters function takes, data-types of parameters and return type of function. By using this information, compiler cross checks function parameters and their data-type with function definition and function call.

Syntax of function prototype :

```
    Return_Type functionName(type1 argument1, type2 argument2, ...);
```

Example :

Let's consider following function definition:

```
int area( int length, int breadth )   //function definition

{

  ....    //function body

}
```

Now, the corresponding prototype declaration of the above function is:

```
   int area( int length, int breadth );    //function prototype
```

It states that function area takes two arguments of type int and returns area of type int.

In the above function prototype ,

- Name of the function is area
- Return type of the function is int.
- Two arguments of type int are passed to the function.

Difference between function prototype and function definition in C

The only difference between the function definition and its function prototype is the addition semicolon (;) at the end of prototype declaration.

But, the parameter identifier could be different in function prototype and function definition because the scope of parameter identifier in a function prototype is limited within the prototype declaration.

Actually, the compiler ignores the name of the parameter list in the function prototype. Having said that, it is good programming practice to include parameter names which increase program clarity.

```
int area(int length, int breadth );

int area(int x, int y );

int area(int , int );
```

All of the above function prototypes are same.

Scope and Conversion of Function Prototype

The scope of the function prototype in C is determined by its position in the program. Generally, the function prototype is placed after the header file in the program. The scope of the function prototype is considered within the same block as the function call.

Calling a function

Control of the program is transferred to the user-defined function by calling it.

Syntax of function call :

```
        functionName(argument1, argument2, ...);
```

In the above example, the function call is made using area(n1, n2); statement inside the main() function.

Function definition

Function definition contains the block of code to perform a specific task.

Syntax of function definition

```
  returnType functionName(type1 argument1, type2 argument2, ...)

  {

     //body of the function

  }
```

When a function is called, the control of the program is transferred to the function definition. And, the compiler starts executing the codes inside the body of a function.

Passing arguments to a function

In programming, argument refers to the variable passed to the function. In the above example, two variables n1 and n2 are passed during the function call.

The parameters length and breadth accepts the passed arguments in the function definition. These arguments are called **formal parameters** of the function.

How to pass arguments to a function ?

```
#include<stdio.h>
int area(int length, int breadth);
void main()
{
……………………..
area_rectangle=area(n1,n2);
…………………….
}
int area(int length, int breadth)
{
```

```
……………………
…………………………
}
```

The type of arguments passed to a function and the formal parameters must match, otherwise, the compiler will throw an error.

Return Statement

The return statement terminates the execution of a function and returns a value to the calling function. The program control is transferred to the calling function after the return statement.
In the above example, the value of the result variable is returned to the main function. The area_rectangle variable in the main() function is assigned this value.

```
#include<stdio.h>
int area(int length, int breadth);

void main()

{

……………………..

area_rectangle=area(n1,n2);

…………………….

}

int area(int length, int breadth)          area_rectangle=result

{

………………

return result;

}
```

The type of value returned from the function and the return type specified in the function prototype and function definition must match.

Types of User-defined functions in C

Based on the arguments and the return type , the function can be classified into the following categories:-

1. Functions with no arguments and no return value.
2. Functions with no arguments and return value.
3. Functions with arguments and no return value.
4. Functions with no arguments and return value.

Function with no arguments and no return value:- When a function has no arguments, it does not receive any data from the calling function. Similarly when it does not return a value, the calling function does not receive any data from the called function.

Syntax :

Function declaration : void function_name();

Function call : function_name();

Function definition :

```
void function_name()

{

  statements;

}
```

We illustrate such function with the help of the following program:-

```
#include <stdio.h>
#include<conio.h>
void main()
{
        clrscr();
void message();
printf("we explain the function with no argument and no return type");
message();
getch();
}

void message()
{
printf("this function neither takes the arguments nor does it return any value"):
}
```

Function with no argument but with a return type:- Such functions don't take any argument but they return the value to the calling function.

Syntax :

Function declaration : int function_name();

Function call : function_name();

Function definition :

```
int function_name()

{
```

```
            statements;

             return x;

          }
```

We give the following program to explain such function:-

```
#include<stdio.h>
#include<conio.h>
void main()
{
clrscr();
int sum();
int result;
result=sum();
printf("the required sum is =%d",result);
getch();
}
int sum()
{
int x,y,sum;
printf("enter the two numbers");
scanf("%d%d",&x,&y);
sum=x+y;
return sum;
}
```

Function with argument but no return type:- Such functions takes the arguments but they do not return any value.

Syntax :

Function declaration : void function_name (int);

Function call : function_name(x);

Function definition:

```
void function_name( int x )

{

     statements;

}
```

We give the following program to explain such function:-

```
#include<stdio.h>
#include<conio.h>
```

```
void main()
{
clrscr();
int x ,y;
void  sum(int,int);                       //function declaration
printf("enter the two numbers");
scanf("%d%d",&x,&y);
sum(x,y);                                           //function calling
getch();
}
void sum(int a,int b)                    //function definition
{
int sum;
sum=a+b;
printf("the required sum is =%d",sum);
}
```

Function with arguments and return type:- Such function do take the arguments and they also return the value to the calling function.

Syntax :

Function declaration : int function_name (int);

Function call : function_name(x);

Function definition:

```
int function_name( int x )

{

  statements;

  return x;

}
```

We explain this function with the following program:-

```
#include<stdio.h>
#include<conio.h>
void main()
{
clrscr();
int x ,y,result;
int add(int,int);                //function declaration
printf("enter the two numbers");
scanf("%d%d",&x,&y);
result = add(x,y);
printf("the required result is=%d",result);          //function calling
getch();
}
int add(int a,int b)                    //function definition
{
int sum;
sum=a+b;
return sum;
}
```

Passing Arguments to a Function

There are different ways in which parameter data can be passed into and out of methods and functions. Let us assume that a function B() is called from another function A(). In this case A is called the "**caller function**" and B is called the "**called function or callee function**". Also, the arguments which A sends to B are called **actual arguments** and the parameters of B are called **formal arguments**.
Formal Parameter : A variable and its type as they appear in the prototype of the function or method.
Actual Parameter : The variable or expression corresponding to a formal parameter that appears in the function or method call in the calling environment.
There are two methods of parameter passing to a function :

1. Call by Value
2. Call by reference

Call by value:- In call by value, the changes made in the formal arguments are not reflected back in the actual arguments. When we pass the parameters to a function using the call by value method then the function creates its own copy of the actual parameters and thus any changes made there are not reflected in the actual arguments.

Call by reference:- In call by reference, the changes made in the formal arguments are reflected back in the actual arguments. When pass the parameters using the call by reference method then the called function does not create its own copy of the original variables rather it refers to the original values. In call by reference, the address of the original parameters is passed to the function. Thus the called function works with the original data and therefore any changes made there are reflected in the actual arguments.

We give the following program to explain call by value and call by reference
/* Swap using call by value */

```
void main()
{
int a=12,b=15;
void swap(int,int);
clrscr();
printf("before calling the swap function \n a=%d and b=%d ",a,b);
swap(a,b);
printf("after calling the swap function \n a=%d and b=%d ",a,b);
getch();
}
void swap(int x, int y )
{
int temp;
temp=x;
x=y;
y=temp;
}
```

The output of this program is as under :

```
before calling the swap function
a=12 and b=15
after calling the swap function
a=12 and b=15
```

/* Swap using call by reference */

```
void main()
{
int a=12,b=15;
void swap(int *,int *);
clrscr();
printf("before calling the swap function \n a=%d and b=%d ",a,b);
swap(&a ,&b);
printf("after calling the swap function \n a=%d and b=%d ",a,b);
getch();
}
void swap(int *x, int *y )
{
int temp;
temp=*x;
*x=*y;
*y=temp;
}
```

The output of this program is as under :

```
before calling the swap function
a=12 and b=15
after calling the swap function
a=15 and b=12
```

Difference between Call by Value and Call by Reference.

Call by Value	Call by Reference
The actual arguments can be variable or constant.	The actual arguments can only be variable.
The values of actual argument are sent to formal argument which are normal variables.	The reference of actual argument are sent to formal argument which are pointer variables.
Any changes made by formal arguments will not reflect to actual arguments.	Any changes made by formal arguments will reflect to actual arguments.